Scheduling non-identical jobs on a batch resource

by

Sebastian Kosch

Supervisor: Prof. J. Christopher Beck

April 2013

Scheduling non-identical jobs on a batch resource

Sebastian Kosch

A thesis submitted in conformity with the requirements
for the degree of *Bachelor of Applied Science*

Supervisor: Prof. J. Christopher Beck, MIE

Division of Engineering Science
University of Toronto
April 2013

Abstract

Ovens, washers, driers and autoclaves are examples of *batch processing machines* and used in many industries. We examine the problem of scheduling jobs of non-identical sizes, processing times and due dates on a single batch machine of finite capacity in a way that minimizes the maximum lateness $L_{\max}$. Previous authors have introduced a branch-and-price method and a new global constraint for constraint programming solvers, both of which perform better than a simple mixed-integer formulation. In this paper, we present four new models: an improved version of the mixed-integer program, a simple constraint programming model, a decomposition approach and a new mixed-integer model using a novel way to compute batch lateness. All models are evaluated on a set of problem instances and compared by performance. We show that the new mixed-integer model rivals the performance of the previously introduced global constraint.

I must acknowledge Professor Chris Beck, one of the best teachers I have had, who took considerable time out from his busy schedule to keep me on track throughout the year.

Thank you for being a model supervisor.

And I could not have made it this far without two incredible parents and an amazing girlfriend.

I love you guys.

Contents

List of Figures

List of Tables

Introduction

This paper discusses four different approaches, and several variations on them, to solving the problem of scheduling non-identical jobs on a batch processing machine. Batch processing machines, for the purposes of this paper, can process multiple, non-identical jobs simultaneously—but all jobs must be loaded into and unloaded from the machine at once, which introduces a considerable twist on the “simple parallel resources” known from typical example problems in existing literature.

The machines in question represents real-life resources like autoclaves or ovens, which can process multiple items at a time, but often cannot be opened at random—in fact, such machines often need to wait for the largest item in the batch to be done before the next batch can be inserted.

Malapert et al. [2012] proposed a global constraint programming (CP) algorithm consisting of a set of filtering rules to solve the problem. He achieved considerably better speeds than with a simple mixed-integer model (MIP), but it seems plausible that the new global constraint is unnecessarily cumbersome to achieve this performance; simple MIP, CP or decomposition approaches are easier to implement and extend. In this paper, we present 1) an improvement to his MIP model, 2) a CP model, 3) a decomposition approach to “divide and conquer” the problem, and 4) a new MIP model that greatly simplifies the original one.

1.1 Problem definition

We describe the problem as follows: assume we are given a set of jobs J, each of which has a processing time (or "length") p_j and a size (or "capacity requirement") s_j. Each job also has a due date d_j. The machine is characterized by its capacity b, and in every batch, the jobs' summed sizes must not exceed this number. All values are integer. 447723 The machine can only process one batch k of jobs at a time, and batches always take as long as the longest job in the batch (i.e. $P_k = \max_{j \in k}(p_j)$). Our objective is to minimize the lateness L of the latest job in J, where L is the difference between the job's completion time C_j and its due date d_j: in formal terms, *min.* $L_{\max} = \max_j(C_j - d_j)$. The job's completion time, however, is the completion time of the batch, which in turn finishes with its *longest* job as stated above.

Malapert uses the standard format established by Graham et al. to summarize the problem as $1|\textit{p-batch}; b < n; \textit{non-identical}|L_{\max}$, where $\textit{p-batch}; b < n$ represents the parallel-batch nature and the finite capacity of the resource. A simpler version with identical job sizes was shown to be strongly NP-hard in [Brucker et al., 1998]; this problem, then, is no less difficult.

It helps to visualize the jobs before delving into the technicalities of scheduling them. Figure 1.1 shows a solution to a sample problem with eight jobs and a resource with capacity $b = 20$.

1.2 Organization of this paper

We first provide some background on the fundamental concepts of MIP and CP (sections ?? and ??), and review some of the most relevant publications on similar batch scheduling problems (section ??) before we describe Malapert's global constraint in some detail (section ??), as well as his original MIP model (section ??).

We then present possible improvements to the model in 4.2. Section 4.3 introduces a CP formulation of the same problem. Sections 4.4 and 4.4.1 describe a decomposition

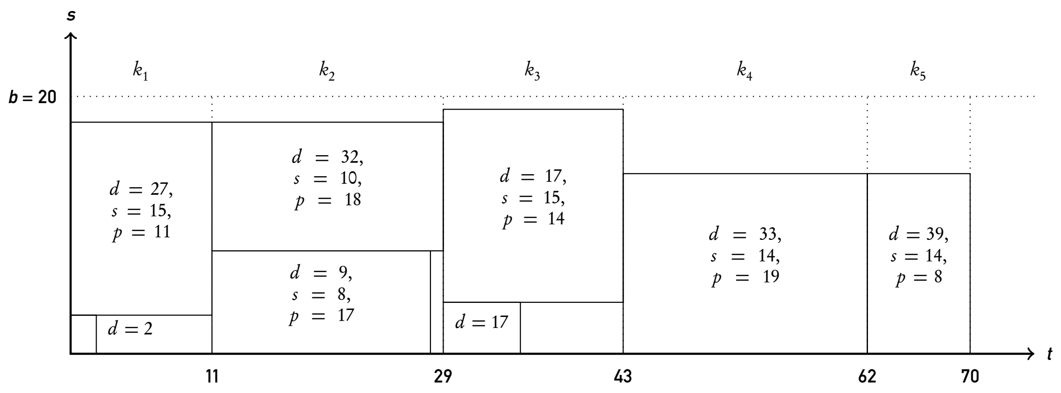


Figure 1.1: Optimal solution to an example problem with eight jobs (s_j and p_j not shown for the two small jobs)

approach. Finally, a new MIP approach is introduced in section **??**.

An empirical comparison of the new models and a discussion of the results follow in sections 5.1 and 5.3. Ideas for future work are listed in 6.

Background

Many optimization problems, and scheduling problems in particular, are combinatorial in nature. The number of possible solutions grows exponentially with the number of input variables, and even with fast computers it is impossible to explore all of them individually to find the best one (also called “full enumeration”, or “brute-force” search) in a reasonable amount of time. Often, however, it is possible to reason about subsets of solutions that are known to be suboptimal a priori. This limits the search space, allowing us to solve many instances of difficult combinatorial problems in few hours, minutes or even seconds.

Such constrained searches are often implemented in either one of two ways (or variants of them): as a *Constraint Programming model* (CP) or as a *Mixed Integer Programming model* (MIP). In this section I will briefly introduce the concepts behind both CP and MIP and review recent work on problems similar to the one dealt with here.

2.1 Constraint Programming

Many problems can be understood as a situation in which a set of values is to be chosen according to certain rules, but since the number of possible combinations of values is enormous, attempting to test all of them is infeasible.

Constraint programming (CP) is a formal framework for the formulation and solution of such problems. CP models consist of a set of variables $V = \{x_1, \ldots, x_n\}$, each x_i of which has a domain $\mathcal{D}_i$, the set of values that could conceivably be assigned to it. The CP problem is defined by a set of relationships that must hold between the variables. Model

2.1.1 illustrates the concept.

$$x \in \{1, \dots, 10\} \tag{2.1}$$
$$y \in \{9, \dots, 20\} \tag{2.2}$$
$$z \in \{1, \dots, 50\} \tag{2.3}$$
$$x > y \tag{2.4}$$
$$z \geqslant xy - 42 \tag{2.5}$$

Model 2.1.1: A simple CP model

Here, x, y, z are the variables with their respective domains.

2.1.1 Propagation

We can choose a variable and a value to assign to it, and explore the consequences of this assignment. Assign $x = 5$, then based on the constraint $x > y$ only values < 5 are permissible for y, but none are available in $\mathcal{D}_y$. The assignment $x = 5$ leads to an *inconsistency*, and another value must be chosen.

If we choose $x = 10$ (which, clearly, is the only feasible value for x, since all others violate $x > y$), y is limited to $y = 9$. This means that $z \geqslant 9 \cdot 10 - 42 = 48$, thus $z = \{48, 49, 50\}$.

This process of successive elimination is known as *propagation*, and it is the core of all CP solver algorithms. In propagation, a variable's domain is reduced, and the constraints are used to reduce the domains of other variables accordingly, until no constraints are violated.

2.1.2 Consistency

Propagation is commonly seen as a way to enforce consistency. It can be triggered by fixing a variable to a value, but also by systematic evaluation of constraints. Let any two variables in a problem be involved in a binary constraint C, such as $x > y$ above. If their domains are reduced such that they fulfill C (that is, $\forall a \exists b$ such that $a \in \mathcal{D}_x, b \in \mathcal{D}_y, C(x = a, y =$

$b) = \texttt{true}$), they are *arc-consistent* with regards to C. Propagation will commonly enforce arc consistency across all variables and constraints.

Arc consistency alone is not sufficient to guarantee the existence of a solution (satisfiability), unless the graph of variables and constraints is acyclic. Consider the following problem (Model 2.1.2):

$$x_1 = x_2 \tag{2.6}$$
$$x_1 = x_3 \tag{2.7}$$
$$x_2 = x_4 \tag{2.8}$$
$$x_3 \neq x_4 \tag{2.9}$$
$$x_1, x_2, x_3, x_4 \in \{1, 2\} \tag{2.10}$$

Model 2.1.2: Unsatisfiable CP problem

The model is arc-consistent but has no solution: forcing any variable to assume a fixed value will result in at least constraint being violated.

The notion of arc consistency (two variables) can be extended, e.g. to three variables related through binary constraints (*path consistency*) or to constraints involving more than two variables, where every value assigned to a variable x must be consistent with all of all other variables' values (*generalized arc consistency*).

2.1.3 Global constraints

Propagation makes use of graph-theory based algorithms that enforce a form of consistency on the variables involved in a constraint. Most constraints are formulated in the form of binary relationships involving a single operator (e.g. $\geqslant, =$ or $\neq$). But it is often possible and beneficial to impose constraints on groups of variables at once. Consider the following classic example: This model prescribes that the variables x_1, x_2, x_3, x_4 are to take on different values. The model is arc consistent. To find a solution (or insatisfiability), a value has to be assigned to three of the variables, and the solver has to propagate the domain reductions after every assignment. Consider, on the other hand, the use of a specialized

$$x_1 \neq x_2 \tag{2.11}$$
$$x_2 \neq x_3 \tag{2.12}$$
$$x_3 \neq x_4 \tag{2.13}$$
$$x_4 \neq x_1 \tag{2.14}$$
$$x_1 \neq x_3 \tag{2.15}$$
$$x_2 \neq x_4 \tag{2.16}$$
$$x_1, x_2, x_3, x_4 \in \{1,2,3\} \tag{2.17}$$

Model 2.1.3: A clique of not-equal constraints

constraint called `alldifferent`(x_1, x_2, x_3, x_4). This constraint, implemented as a custom algorithm, can speed up the propagation rapidly. In the case of our example, the constraint can infer that the number of variables exceeds the number of available values. The solver can conclude that the problem is insatisfiable without any propagation.

Such specialized or "global" constraints can efficiently exploit properties that hold for certain relationships among groups of variables, and that would not be available if the relationship were expressed as a set of binary constraints.

2.2 Mixed Integer Programming

2.2.1 Linear Programming

Mixed Integer Programs (or "MIP models") express the minimization of a linear function subject to linear constraints. If all variables in the problem can be rational in the solution, the MIP model is really a *linear program* (LP), which can be solved in polynomial time.[1]

Figure 2.1 illustrates the concept of an LP in two variables, x_1 and x_2, as listed in model 2.2.1. The set of feasible solutions is given by the shaded area bounded by the axes and by three inequalities ("constraints"). A third linear term, the objective function $3x_1 + x_2$, is to

[1]In practice, variations on Dantzig's *simplex method* are most often used to solve LPs. Such solvers perform very well on most problems, but no known variant has been proven to have polynomial worst-case complexity [Papadimitrou and Steiglitz, 1998]. Solvers with theoretically polynomial-time complexity exist (Karmarkar's algorithm Karmarkar [1984] has a runtime of $\mathcal{O}(n^{3.5}L^2 \cdot \log L \cdot \log\log L)$, for instance, where L is the number of bits of input), but are used less frequently.

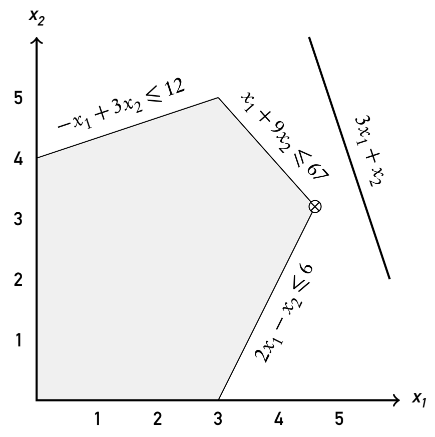


Figure 2.1: Graphical representation of a typical LP problem

be maximized.[2]

$$\begin{aligned}
\text{Maximize} \quad & 3x_1 + x_2 && (2.18)\\
\text{subject to the constraints} \quad & -x_1 + 3x_2 \leqslant 12 && (2.19)\\
& x_1 + 9x_2 \leqslant 67 && (2.20)\\
& 2x_1 - x_2 \leqslant 6 && (2.21)\\
& x_1 \geqslant 0 && (2.22)\\
& x_2 \geqslant 0 && (2.23)
\end{aligned}$$

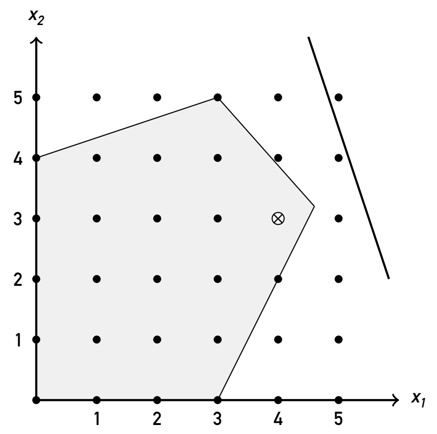
Model 2.2.1: A simple LP model, as shown in figure 2.1

Although the objective function is shown in the figure as a line in a specific location, note that, as it is not an equation, it can be represented by any line parallel to that shown in the figure. While we are trying to maximize the objective value, the solution must be feasible. It is thus obvious that the desired extreme value of our objective function is found at one of the "corners" of the shaded area: the solution is marked ⊗ in the figure. In fact, since the shaded area generated by linear inequalities will always be a convex polygon (or polyhedron, in higher dimensions), the solution will invariably be found at an intersection of hyperplanes.

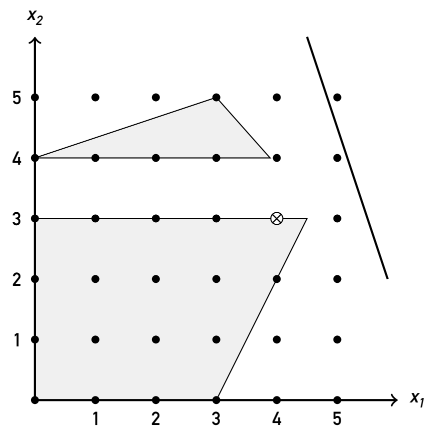
[2]Minimization is more common, but note that multiplying the objective by −1 achieves this.

2.2.2 Solving MIP models using branch-and-bound

MIP models are LPs in which some of the decision variables are declared integers—thus their name. Like LP models, MIP models also require an objective function to be minimized. Figure 2.2a illustrates this based on the LP problem above: now, only the black dots



(a) at the root node



(b) before first branching

Figure 2.2: Graphical representation of a typical MIP problem

represent feasible solutions. While the solution is easily found in the figure by inspection (simply round to the nearest feasible integral solution!), this is not the case in problems with many variables; it is difficult enough to visualize the problem in three dimensions, and many problems require hundreds or thousands. Moreover, rounding is particuarly unreliable with variables of small domains (e.g. binary variables), which are often used in MIP models to represent decisions. Indeed, solving MIP models is NP-hard.

Similar to CP searches, MIP searches can be thought of as trees, where every branch represents an assignment of a value to a domain and every leaf represents a feasible solution. The difference lies in the way MIP explores this search tree.

The MIP solver first solves the problem as an LP, which will usually result in fractional values for all or most of the variables. In this context, the LP is known as the *LP relaxation* of the problem—an easier, approximate version of the original. Assuming we are minimizing the objective, the resulting LP objective value will be a lower bound on the optimal

MIP objective value. At this point, the solution is $x_1 = 4.6, x_2 = 3.2$.

The solver then chooses a variable, based on heuristics, and *branches* on it. In this case, assume that we branch on x_2, which means that we set $x_2 \leqslant 3.0 \lor x_2 \leqslant 4.0$. We now have two new MIPs, as shown in figure 2.2b. Both of these new subproblems are need solving.[3] At this point, another heuristic decides which subproblem is solved first. Again, the chosen subproblem is first solved in its LP relaxation, and the above procedure is repeated until an integral solution is found. If this solution is the best solution known so far, it is stored as the *incumbent*. The solver then *backtracks*: it undoes some of the previously fixed variables, and explores other (previously unsolved) subproblems.

This behaviour can be visualized as a search tree in which every node represents a decision between two subproblems. At every node, a subproblem is chosen, the subproblem's LP relaxation is run, and the next variable is chosen to be branched on.

When the solver backtracks to a previous node to explore another subproblem, say P_1, it can compare the LP solution of P_1 with the current incumbent's solution. Since a solution will never improve with additional integrality constraints, the subtree of P_1 can be ignored if the objective value of the LP solution to P_1 is worse than that of the current incumbent. This "pruning" of the search tree during the search is referred to as *bounding*, and is what allows many MIP models to be solved in less time than their theoretically exponential complexity suggests.

2.2.3 Lazy constraints in MIP

Achieving fast search times in MIP requires striking a delicate balance between keeping the model small (reducing the time needed to solve the LP relaxations) and formulating the model tightly (pruning more branches off the search tree). Often, the way certain constraints can be used to perform pre-solve reductions plays into the performance also, and is difficult to predict.

[3]In this example, the optimal value for x_2 was within ± 1 of its LP value. This is often the case, and it is not immediately obvious from the figure why fixing $x_2 = 3.0 \lor x_2 = 4.0$ is insufficient. In some problems, however, the circumscribed polytope protrudes relatively far diagonally through the grid of integer solutions, in which case branching on $\leqslant \lfloor x_i \rfloor$ and $\geqslant \lceil x_i \rceil$ is necessary to capture the optimal feasible solution.

Lazy constraints are one way to avoid this problem. They are expressed as regular linear constraints and checked against whenever the model encounters a new solution. Lazy constraints that were violated are then added to the model to be used in future instances.

This is particularly useful when a problem lends itself to the generation of vast sets of constraints involving few variables, but most of these constraints are likely not needed. In such cases, lazy constraints can greatly speed up solution times.

2.3 Scheduling using CP and MIP

2.3.1 Common characteristics of scheduling problems

Many scheduling problems can be described by some of the following characteristics:

Objective In this paper, $L_{\max}$ is the value to be minimized. Other situations might call for minimization of the makespan ($C_{\max}$), the maximum tardiness (like lateness, but never below zero), total completion time weighed differently for every job, or other measures.

Number of resources Some problems, like the one treated in this paper, assume the existence of only one resource. Other problems, e.g. *flow shop* or *open shop* problems, require multiple resources.

Release dates and precendence constraints Many problems specify that certain jobs must finish before other jobs begin, and/or that they may not start before a certain time.

Resource type In this paper, the resource processes batches only. Other resources may be *disjunctive*, allowing execution of one job at a time only, or *cumulative*, allowing for multiple jobs to be processed in parallel without any batching restriction. *Preemptive* processing means that jobs may be interrupted and continued at a later time, which greatly improves the flexibility and, in turn, the objective value of the resulting schedule.

Scheduling problems are commonly summarized as an abbreviated expression summarizing the above and other properties in a format introduced in Graham et al. [1979]. For the problem at hand, it is $1|\textit{p-batch}; b < n; \textit{non-identical}|L_{\max}$.

2.3.2 Scheduling problems in CP

CP solvers work by reducing the domains of variables, and so it is only natural to express the position of a job using variables such as `startOf`(j) or `endOf`(j), the domains of which are then reduced to feasible values. This creates the notion of an earliest/latest start/end time of a job, commonly abbreviated `est`/`lst`/`eft`/`lft`.

The most well-known propagation techniques on these four variable bounds are called *edge finding*: they reason about a job's time bounds and whether they allow it (or force it) to execute before or after a set of other jobs on the same resource.

Capacity constraints are often expressed as `cumulative` and/or `pack` global constraints, efficient (and problem-agnostic) implementations of which are part of every major CP software package.

2.3.3 Scheduling problems in MIP

The position of a job in time is more difficult to represent in MIP formulations. Most models can roughly be divided into two groups:

Time-indexed/discrete-time formulations. In this type of model, a set of binary variables is used to indicate the start time (or end time) of a job on a predefined, discrete grid of time points. The binary variables must sum to one. Capacity constraints (e.g. cumulative constraints) can then be imposed by means of relationships between start times of pairs of jobs, their sizes and processing times. This kind of model is be relatively easy to extend and work with, but the number of variables and constraints required grows (at least) with the square of the number of jobs involved.

Disjunctive/continuous-time formulations. In this type of model, jobs' start times are represented by single variables, and for a pair of jobs, the difference between their start times must exceed the earlier-scheduled job's processing time on a disjunctive resource.

2.4 Literature Review

An extensive and application-based introduction to MIP models is given by Williams [1978]

The problem at hand is based on the work of Malapert et al. [2012], who proposed a global constraint `sequenceEDD` to be used in combination with `pack` to solve it to optimality. The `sequenceEDD` constraint is implemented as four distinct filtering rules applied at relevant domain changes: three to update bounds on $L_{\max}$ based on different conditions, and one to limit the number of batches based on the marginal cost difference between adding a job to an empty batch vs. an existing one. The new global was shown to significantly outperform a simple MIP model of the same problem, as well as a branch-and-price model proposed by Daste et al. [2008].

Other authors have examined similar problems: Azizoglu and Webster [2000] provide an exact method and a heuristic for the same problem, but minimize makespan ($C_{\max}$) instead of $L_{\max}$, as have Dupont and Dhaenens-Flipo [2002]. Similar exact methods have been proposed for multi-agent variants with different objective functions [Sabouni and Jolai, 2010], for makespan minimization on single batch machines [Kashan et al., 2009], and for makespan minimization on parallel batch machines with different release dates [Ozturk et al., 2012]. A more extensive review of MIP model applications in batch processing is given by Grossmann [1992].

State of the Art: Recent work by Malapert

3.1 The sequenceEDD global constraint

In their 2011 paper, Malapert et al. present a CP formulation of the problem which, in essence, relies on two global constraints: `pack`, which constraints job-to-batch assignments such that no capacity limits are violated, and `sequenceEDD`, which sorts batches into EDD order. The implementation of the latter constraint includes a set of rules which update the current $L_{\max}$, lower and upper bounds on $L_{\max}$, and lower and upper bounds on the number of batches in the optimal solution every time `sequenceEDD` is triggered by a job-batch assignment. Based on these bounds, other assignments are then eliminated from the set of feasible assignments.

3.1.1 Filtering rules

***Final lateness filtering rule* and *lateness filtering rule*.** The algorithm calculates the value of $L_{\max}$ given a partial assignment $\mathcal{A}$. This value is also a lower bound on the $L_{\max}$ of any assignment $\mathcal{A}'$ extending $\mathcal{A}$, since assigning more jobs to batches will never improve $L_{\max}$.[1]

Cost-based domain filtering rule of assignments. This rule eliminates possible assignments of yet-unassigned jobs j to batches k, if such assignments cause $L_{\max}$ to exceed the

[1]An *assingment* $\mathcal{A}$ in this context refers to a set of job-to-batch assignments for any subset of jobs $J_{\mathcal{A}} \subseteq J$. A given assignment $\mathcal{A}$ can be *extended* to $\mathcal{A}'$ by assigning one or more additional (i.e. previously unassigned) jobs to batches.

current known upper bound on $L_{\max}$ (or best known solution).

Cost-based domain filtering based on bin packing. The following set of rules considers the number of non-empty batches M used in the final solution. First, lower and upper bounds on M are updated based on the number of batches currently in use and on the number of unassigned ("open") jobs. Then, the set of shortest remaining jobs is used to update to compute a lower bound on $L_{\max}$ given a hypothetical solution using exactly $M = m$ non-empty batches. Based on this lower bound, the current lower bound on $L_{\max}$ is updated. Finally, the upper bound on M is decremented until the lower bound on $L_{\max}$ given M batches no longer exceeds the current upper bound on $L_{\max}$.

These four filtering rules serve to systematically reduce the solution space as the search progresses.

3.1.2 Search heuristic

The search heuristic chosen for the packing of a job j into a set of batches K is to prefer assignment to batch k having the lowest value of $\gamma(B_j \leftarrow k)$, approximating the fitness of j into k given the current assignment $\mathcal{A}$:

$$\gamma(B_j \leftarrow k) = \frac{|\min(P_k) - p_j|}{\max_J(p_j) - \min_J(p_j)} + \frac{|\max(D_k) - d_j|}{\max_J(d_j) - \min_J(d_j)} \tag{3.1}$$

We reference this heuristic in section B.2.1, where we use it to determine an initial upper bound on the value of $L_{\max}$.

3.2 MIP formulation

Malapert's original MIP formulation is given in Model 3.2.1. It uses a set of binary decision variables x_{jk} to represent whether job j is assigned to batch k. The original model assumes a set K of $|K| = |J|$ batches, the number of jobs being a trivial upper bound on the number of batches required; it also enforces an earliest-due-date-first (EDD) ordering of the batches (constraint (3.8)).

$$
\begin{array}{llll}
\text{Min.} & L_{\max} & & (3.2)\\
\text{s.t.} & \sum_{k\in K} x_{jk} = 1 & \forall j \in J & (3.3)\\
& \sum_{j\in J} s_j x_{jk} \leqslant b & \forall k \in K & (3.4)\\
& p_j x_{jk} \leqslant P_k & \forall j \in J, \forall k \in K & (3.5)\\
& C_{k-1} + P_k = C_k & \forall k \in K & (3.6)\\
& (d_{max} - d_j)(1 - x_{jk}) + d_j \geqslant D_k & \forall j \in J, \forall k \in K & (3.7)\\
& D_{k-1} \leqslant D_k & \forall k \in K & (3.8)\\
& C_k - D_k \leqslant L_{\max} & \forall k \in K & (3.9)\\
& C_k \geqslant 0, P_k \geqslant 0 \text{ and } D_k \geqslant 0 & \forall k \in K & (3.10)
\end{array}
$$

Model 3.2.1: Malapert's original MIP model

Constraints (3.3) ensure that every job j is assigned to one batch k, and one batch only. Constraints (3.4) ensure that no batch exceeds the machine's capacity b. Constraints (3.5) define each batch's processing time P_k as the maximum processing time of the jobs j assinged to it. Constraints (3.6) define each batch's completion time C_k as that of the previous batch, plus the previous batch's processing time. Constraints (3.8) ensure that batches are ordered by due date. Constraints (3.9) define the objective value $L_{\max}$.

Modelling the problem

In this section we present the models examined. Beginning with the MIP model used by Malapert, we explore several additional constraints that, while redundant, tighten the search space. We comment on the performance of the approaches (see section **??** for a comprehensive comparison of performance). We present a CP model and a decomposition-based approach using either MIP or CP. Finally, we introduce a new approach to setting up a MIP model that significantly improves upon the original MIP and outperforms all other models presented in this paper.

All MIP and CP models were implemented in C++ using IBM Ilog's Cplex and CP Optimizer solvers, respectively.

4.1 MIP model

We first replicated Malapert's original MIP approach, as given in Model 3.2.1, to establish a performance baseline to compare other models to. The original model assumes a set K of $|K| = |J|$ batches, the number of jobs being a trivial upper bound on the number of batches required; it also enforces an earliest-due-date-first (EDD) ordering of the batches (constraint (3.8) in Model 3.2.1).

4.2 Improved MIP model

Several improvements can be made to Malapert's MIP model in the form of additional constraints shown in Model 4.2.1, as described in greater detail in the subsections below.

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K \tag{4.1}$$

$$e_k + \sum_{j\in J} x_{jk} \geqslant 1 \quad \forall k \in K \tag{4.2}$$

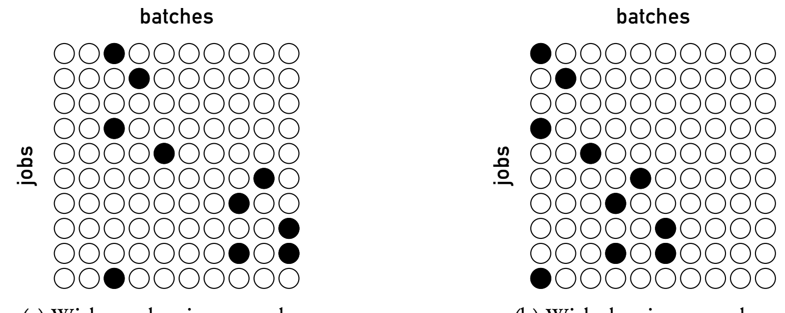
$$n_j(e_k - 1) + \sum_{j\in J} x_{jk} \leqslant 0 \quad \forall k \in K \tag{4.3}$$

$$e_k - e_{k-1} \geqslant 0 \quad \forall k \in K \tag{4.4}$$

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{4.5}$$

$$L_{\max} \geqslant \lceil \frac{1}{b} \sum_j s_j p_j \rceil - \delta_q \quad \forall q, \forall \{j \in J | d_j \leqslant \delta_q\} \tag{4.6}$$

Model 4.2.1: Improvements to Malapert's original MIP model



(a) Without dominance rule (b) With dominance rule

Figure 4.1: Dominance rule to eliminate empty batches followed by non-empty batches (circles represent the x_{jk} variables; a filled circle stands for $x_{jk} = 1$)

4.2.1 Grouping empty batches

The given formulation lacks a rule that ensures that no empty batch is followed by a non-empty batch. Empty batches have no processing time and a due date only bounded by $d_{\max}$, so they can be sequenced between non-empty batches without negatively affecting $L_{\max}$. Since, however, desirable schedules have no empty batches scattered throughout, we can easily reduce the search space by disallowing such arrangements. The idea is illustrated in Figure 4.1.

A mathematical formulation is

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K. \tag{4.7}$$

To implement this, we can write constraints in terms of an additional set of binary variables, e_k, indicating whether a batch k is empty or not:

$$e_k + \sum_{j\in J} x_{jk} \geqslant 1 \qquad \forall k \in K, \tag{4.8}$$

$$n_j(e_k - 1) + \sum_{j\in J} x_{jk} \leqslant 0 \quad \forall k \in K. \tag{4.9}$$

Constraints (4.8) enforce $e_k = 1$ when the batch k is empty. Constraints (4.9) enforce $e_k = 0$ otherwise, since the sum term will never exceed the number of jobs n_j. The rule (4.7) can now be expressed as $e_{k-1} = 1 \rightarrow e_k = 1$, and implemented as follows:

$$e_k - e_{k-1} \geqslant 0 \quad \forall k \in K. \tag{4.10}$$

We can also prune any attempts to leave the first batch empty by adding a constraint $e_0 = 0$.

4.2.2 No postponing of jobs to later batches

Consider a schedule in which all jobs are assigned to a single batch (i.e. $x_{jk} = 1 \forall j = k$), ordered by non-decreasing due date, and by non-decreasing processing time in case of a tie between two jobs. For the purposes of this paper, we refer to this schedule as the *single-EDD* schedule.

Since, as shown above, there exists an optimal solution with EDD-ordered batches, we can restrict the search to such solutions. Any solution in which a job j is moved into a batch h later than its single-EDD batch k, i.e. $h > j = k$, necessarily results in a non-EDD solution. We can therefore write

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{4.11}$$

to exclude solutions in which jobs are assigned to later batches than their respective single-EDD batch.

4.2.3 Lower bound on $L_{\max}$

Let a *bucket* q denote the set of all batches with due date δ_q. Then the completion date C_q of this bucket is the completion date of the last-scheduled batch with due date δ_q, and the lateness of the bucket q is $L_q = C_q - \delta_q$. Since all batches up to and including those in bucket q are guaranteed to contain all jobs with due dates $d \leqslant \delta_q$, as ensured by the EDD ordering of batches, the lower bound on every bucket's lateness $LB(L_q)$ is a valid lower bound on $L_{\max}$. In other words, jobs with due date $d \leqslant \delta_q$ will be found only in batches up to and including the last batch of bucket q. This provides a lower bound on the lateness of bucket q:

$$L_{\max} \geqslant C_{\max,q} - \delta_q \quad \forall q \tag{4.12}$$

The buckets up to bucket q will likely also contain some later ($d > \delta_q$) jobs in the optimal solution, which weakens $LB(L_q)$'s usefulness but does not affect its validity as a lower bound.

Now we need to find $C_{\max,q}$, or at least a lower bound on it, in polynomial time. The simplest approach simply considers the jobs' total "area" (or "energy"), i.e. the sum of all $s_j p_j$ products:

$$C_{\max,q} \geqslant \left\lceil \frac{1}{b} \sum_j s_j p_j \right\rceil \quad \forall q, \forall \{j \in J | d_j \leqslant \delta_q\} \tag{4.13}$$

A better lower bound on $C_{\max,q}$ would be given by a preemptive-cumulative schedule. Unfortunately, minimizing $C_{\max}$ for such problems is equivalent to solving a standard bin-packing problem, which requires exponential time.[1]

[1] In a preemptive-cumulative schedule, jobs may be stopped and restarted mid-execution, but occupy a constant amount s_j on the resource while executing. In such a schedule, minimizing the makespan is as difficult as solving a bin-packing problem: we can break jobs into small pieces (no longer than the smallest common divisor of the jobs' lengths p) and then pack them together such as to minimize the number of small time slots needed.

4.3 CP model

The constraint programming model is based on a set of decision variables $B_j = \{k_1, \ldots, k_{n_k}\}$, where each variable stands for the batch job j is assigned to. The complete model is given in Model 4.3.1.

$$\begin{aligned}
\text{Min.} \quad & L_{\max} && && (4.14)\\
\text{s.t.} \quad & \texttt{pack}(J, K, b) && && (4.15)\\
& \texttt{cumul}(J, b) && && (4.16)\\
& P_k = \max_j p_j && \forall \{j \in J | B_j = k\}, \forall k \in K && (4.17)\\
& D_k = \min_j d_j && \forall \{j \in J | B_j = k\}, \forall k \in K && (4.18)\\
& C_k + P_{k+1} = C_{k+1} && \forall k \in K && (4.19)\\
& L_{\max} \geqslant \max_k (C_k - D_k) && && (4.20)\\
& \texttt{IfThen}(P_k = 0, P_{k+1} = 0) && \forall k \in \{k_1, \ldots, k_{n_k - 2}\} && (4.21)\\
& B_j \leqslant k && \forall \{j \in J, k \in K | j > k\} && (4.22)
\end{aligned}$$

Model 4.3.1: Constraint programming model

Constraint (4.15) makes sure the jobs are distributed into the batches such that no batch exceeds the capacity b. Constraint (4.16), a global cumulative constraint, keeps jobs from overlapping on the resource—while redundant with (4.15), this speeds up propagation slightly. Constraints (4.17), (4.18), (4.19) and (4.20) define P_k, D_k, C_k (batch completion time) and $L_{\max}$, respectively.

4.3.1 Grouping empty batches

We can force empty batches to the back and thus establish dominance of certain solutions. The implementation is much easier than in the MIP model:

$$\texttt{IfThen}(P_k = 0, P_{k+1} = 0) \quad \forall k \in \{k_1, \ldots, k_{n_k - 1}\} \qquad (4.23)$$

4.3.2 No postponing of jobs to later batches

Just like in the MIP model, jobs should never go into a batch with an index greater than their own:

$$B_j \leqslant k \quad \forall \{j \in J, k \in K | j > k\} \tag{4.24}$$

4.4 Decomposition approach

Instead of solving the entire problem using one model, we can solve the problem step by step, using the best techniques available for each subproblem. This approach is inspired by *Benders decomposition*.

A basic version of this approach uses branch-and-bound to transverse the search tree. At each node on level ℓ, a single MIP and/or CP model is run to assign jobs to batch $k = \ell$. The remaining jobs are passed to the children nodes, which assign jobs to the next batch, and so on—until a solution, and thus a new upper bound on $L_{\max}$, is found. Several constraints are used to prune parts of the search tree that are known to offer only solutions worse than this upper bound. Figure 4.2 shows an example in which a MIP model is used to assign jobs to the batch at every node. Algorithm 1 outlines what happens at each node:

Algorithm 1 MIP node class code overview

```
update currentAssignments                                  ▷ this keeps track of where we are in the tree
if no jobs given to this node then                         ▷ if this is a leaf node, i.e. all jobs are assigned to batches
    calculate $L_{\max,\text{current}}$ based on currentAssignments
    if $L_{\max,\text{current}} < L_{\max,\text{incumbent}}$ then
        $L_{\max,\text{incumbent}} \leftarrow L_{\max,\text{current}}$
        bestAssignments ← currentAssignments
    end if
    return to parent node
end if
set up MIP model                                           ▷ as described below
repeat
    $x_j \leftarrow$ model.solve($x_j$)                    ▷ let model assign jobs to the batch
    spawn and run child node with all $\{j | x_j = 0\}$    ▷ pass unassigned jobs to children
    add constraint to keep this solution from recurring    ▷ this happens once the child node returns
until model has no more solutions
return to parent node
```

$$\begin{tikzcd}
 & & & \text{MIP} \arrow[dll] \arrow[d] \arrow[drr] & & & & k=1 \\
 & \text{MIP} \arrow[dl] \arrow[dr] & & \text{MIP} & & \text{MIP} \arrow[dl] \arrow[dr] & & k=2 \\
\text{MIP} & & \text{MIP} \arrow[d] & & \text{MIP} \arrow[d] & & \text{MIP} & k=3 \\
 & & \text{Solution 1} & & \text{Solution 2} & & &
\end{tikzcd}$$

Figure 4.2: Batch-by-batch decomposition using MIP only

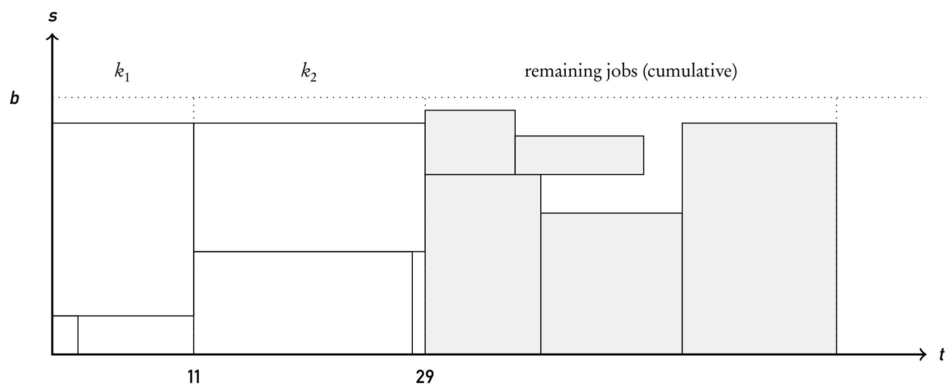


Figure 4.3: Constructing the second batch. Remaining jobs are shaded.

the model finds the best jobs to assign to the batch according to some rule, lets the children handle the remaining jobs, and tries the next best solution once the first child has explored its subtree and backtracked.

Using MIP and cumulative packing after the batch

The first version of the branch-and-bound batch-by-batch decomposition uses a MIP model at each node to assign jobs to the respective batch. The remaining jobs are packed such as to minimize their $L_{\max}$, with a relaxation of the batching requirement, i.e., as if on a cumulative resource as illustrated in Figure 4.3.

Model 4.4.1 implements a time-indexed cumulative constraint on the non-batched jobs. Constraints (4.26) through (4.28) ensure that the batch stays below capacity, define the duration of the batch P_k and force at least one of the earliest-due jobs into the batch.

Constraints (4.29) express the interval relaxation as follows: for any job j, the sum of

$$
\begin{array}{llll}
\text{Min.} & L_{\text{max,cumul}} & & (4.25) \\
\text{s. t.} & \sum_j s_j x_j \leqslant b & \forall j \in J & (4.26) \\
& P_k \geqslant p_j x_j & \forall j \in J & (4.27) \\
& \sum_j x_j \geqslant 1 & \forall \{j \in J | d_j = \min(d_j)\} & (4.28) \\
& P_k + \frac{1}{b}\sum_i (1 - x_i) s_i p_i \leqslant d_j + L_{\text{max,incmb}} - 1 - v_k & \forall j \in J, \forall \{i \in J | d_i \leqslant d_j\} & (4.29) \\
& \sum_t u_{jt} = 1 & \forall j \in J & (4.30) \\
& \sum_j \sum_{t' \in T_{jt}} s_j u_{jt'} \leqslant b & \forall t \in \mathcal{H} & (4.31) \\
& (v_k + t + p_j) u_{jt} \leqslant d_j + L_{\text{max,incmb}} - 1 & \forall j \in J, \forall t \in \mathcal{H} & (4.32) \\
& L_{\text{max,cumul}} \geqslant (v_k + t + p_j) u_{jt} - d_j & \forall j \in J, \forall t \in \mathcal{H} & (4.33) \\
& u_{j,t=0} = x_j & \forall j \in J & (4.34) \\
& u_{it} \leqslant (1 - x_j) & \forall i, j \in J, \forall t \in \{1, \dots, p_j - 1\} & (4.35) \\
& b - \sum_{i \in J} s_i x_i \leqslant (b w_j + 1) s_j & \forall j \in J & (4.36) \\
& P_k + 2 w_j n_t \geqslant p_j + n_t x_j & \forall j \in J & (4.37) \\
& P_k - 2(1 - w_j) n_t \leqslant p_j + n_t x_j - 1 & \forall j \in J & (4.38)
\end{array}
$$

Model 4.4.1: MIP model in batch-by-batch branch-and-bound

P_k, the processing time of the batch, and the area-relaxed processing time approximation of all non-batched jobs earlier than and including j, must not exceed the latest finish time of j. Even jobs that are assigned to the batch have to fulfill this requirement.

Constraints (4.30) and (4.31) implement the cumulative nature of the post-batch assignments by ensuring that each job starts only once, and no jobs overlap on a given resource at any time.

x_j	is 1 iff job j is assigned to the batch
u_{jt}	is 1 iff job j starts in time slot t
T_{jt}	is the set of all time slots occupied by job j if it ended at time t, that is $T_{jt} = \{t - p_j + 1, \ldots, t\}$
v_k	is the start time of the batch at the given node in the search tree
$L_{\max,\text{incmb}}$	is the incumbent (known best) value of and thus an upper bound on $L_{\max}$
$\mathcal{H}$	is the set of all indexed time points $\{0, \ldots, n_t - 1\}$
w_j	is 1 iff job j is either longer than the batch ($p_j > P_k$) or already part of the batch ($x_j = 1$). Neither condition must be fulfilled for constraint (4.36) to have an effect on job j

Table 4.1: Notation used in the decomposition model

Constraints (4.32) again limit the possible end dates of a job, but unlike (4.29), they use the time assignments on the cumulative resource to determine end dates. Constraints (4.33) define the value of $L_{\max,\text{cumul}}$, the maximum lateness of any job in the non-batched set.

Constraints (4.34) force batched jobs to start at $t = 0$, while (4.35) force *all* jobs to start either at $t = 0$ or after the last batched job ends.

Constraints (4.36) enforce a dominance rule: jobs must be assigned to the batch such that the remaining capacity, $b_r = b - \sum_j s_j x_j$, is less than the size s_j of the *smallest* job from the set of non-batched jobs that are *not longer* than the current batch. That is, if there exists an non-batched job j with $p_j \leqslant P_k$ and $s_j \leqslant b_r$, then the current assignment of jobs is infeasible in the model. The reasoning goes as follows: given any feasible schedule, assume there is a batch k_a with b_r remaining capacity and a later batch k_b containing a job j such that $s_j \leqslant b_r$ and $p_j \leqslant P_{k_a}$. Then job j can always be moved to batch k_a without negatively affecting the quality of the solution: if the schedule was optimal, then moving j will not affect $L_{\max}$ at all (otherwise, it was no optimal schedule); if the schedule was not optimal, then $L_{\max}$ will stay constant (unless j was the longest job in k_b and $L_{\max}$ occured in or in a batch after k_b, in which case $L_{\max}$ will be improved).

This rule is implemented by means of a binary variable w_j, which, as defined by constraints (4.36) and (4.37), is 1 iff $p_j > P_k \vee x_j = 1$. These are the cases in which a job j is *not* to be considered in (4.35), and so w_j is used to scale s_j to a value insignificantly large in the eyes of the constraint's less-than relation.

After a solution is found, a child node in the search tree has run the subtree and returned, a constraint of the form

$$\sum_j x_j + \sum_i (1 - x_i) \leqslant n_j - 1 \quad \forall\{j \in J | x_j = 1\}, \forall\{i \in J | x_i = 0\} \tag{4.39}$$

is added to the model before the solver is called again, to exclude the last solution from the set of feasible solutions.

4.4.1 Using CP and cumulative packing after the batch

This approach is equivalent, but we now use CP to select the batch assignments, again based on a minimized $L_{\max}$ among the non-batched jobs. Model 4.4.2 uses interval variables j to represent the jobs; time constraints on the jobs ("est", "lft" etc.) are represented as functions such as `startOf`(j) and `endOf`(j).

Contraints (4.41) through (4.44) bi-directionally define the relationship between a job's x_j and `startOf`(j): $x_j = 1$ is equivalent with a start time of $t = 0$, and $x_j = 0$ is equivalent with a start time of $t \geqslant P_k$. The term $\sum_{i \in J} p_i$ is used as a large constant as it is greater than any job's start time.

Constraint (4.48) is equivalent to constraints (4.36) through (4.38) in Model 4.4.1 above. The cumulative constraint (4.51) ensures that jobs do not overlap.

4.5 Move-based MIP model

This approach is based on the idea of preserving EDD ordering among the batches. Let an initial schedule be the single-EDD schedule as defined in section 4.2.2. We can calculate the lateness $L_{k,\text{single}}$ of every job in this schedule in linear time.

For the purpose of this discussion, we use the following terminology: in any batch k holding multiple jobs in an EDD schedule, let *host job* denote the earliest-due job in the

$$
\begin{aligned}
\text{Min.}\quad & L_{\max} && && (4.40)\\
\text{s.t.}\quad & \texttt{startOf}(j) \geqslant (1-x_j)P_k && \forall j \in J && (4.41)\\
& \texttt{startOf}(j) \leqslant (1-x_j)\sum_{i\in J} p_i && \forall j \in J && (4.42)\\
& x_j \geqslant \frac{\frac{1}{2} - \texttt{startOf}(j)}{\sum_{i\in J} p_i} && \forall j \in J && (4.43)\\
& x_j \leqslant 1 + \frac{\frac{1}{2} - \texttt{startOf}(j)}{\sum_{i\in J} p_i} && \forall j \in J && (4.44)\\
& P_k \geqslant \max_{j\in J}(x_j p_j) && && (4.45)\\
& \texttt{endOf}(j) = d_j + L_{\max,\text{incmb}} - 1 && \forall j \in J && (4.46)\\
& L_{\max} \geqslant \max_{j\in J}(\texttt{endOf}(j) - d_j) && && (4.47)\\
& \texttt{IfThen}(p_j \leqslant P_k \wedge x_j = 0, b - \sum_{j\in J} s_j x_j \leqslant s_j) && \forall j \in J && (4.48)\\
& P_k + \frac{1}{b}\sum_i s_i p_i \leqslant d_j + L_{\max,\text{incmb}} - 1 - v_k && \forall j \in J, \forall \{i \in J | d_i \leqslant d_j\} && (4.49)\\
& \sum_j x_j \geqslant 1 && \forall \{j \in J | d_j = \min(d_j)\} && (4.50)\\
& \texttt{cumul}(J, b) && && (4.51)
\end{aligned}
$$

Model 4.4.2: CP model in batch-by-batch branch-and-bound

batch, and *guest jobs* all other jobs. If a batch k only holds one job j, then job j is said to be *single*.

We now observe the following two things:

Proposition 1. *Consider a schedule in which batches are ordered by EDD. Then moving job j from batch k_β into an earlier batch k_α will preserve EDD ordering if j is single in k_β and if k_α is not empty.*

Proof. Moving j into k_α will leave D_α unaffected, since k_α is not empty, i.e. there is a host job in k_α. Since the original schedule was EDD-ordered, the host in k_α is due earlier than j. Note that if k_α is empty, the move will potentially disrupt EDD ordering, and is thus prohibited.

Batch k_β will be reduced to zero processing time, but batches after k_β will still be due after $k_{\beta-1}$, so EDD ordering is preserved. □

Proposition 2. *Starting from a single-EDD schedule, moving single jobs back into earlier non-empty batches will generate all possible EDD schedules, with the exception of those EDD schedules in which two hosts of the same due date are sorted in order of decreasing processing time.*

Proof. The order in which moves are performed is arbitrary; moves are independent of each other (capacity requirements notwithstanding). Therefore, any job can be moved into any earlier bucket (i.e. set of batches of equal due date). Within buckets, jobs are free to move with the only condition that host jobs be sorted by non-decreasing processing time, as given by the intial single-EDD schedule. Under this condition, then, all possible EDD schedules can be generated by moving single jobs into earlier batches. □

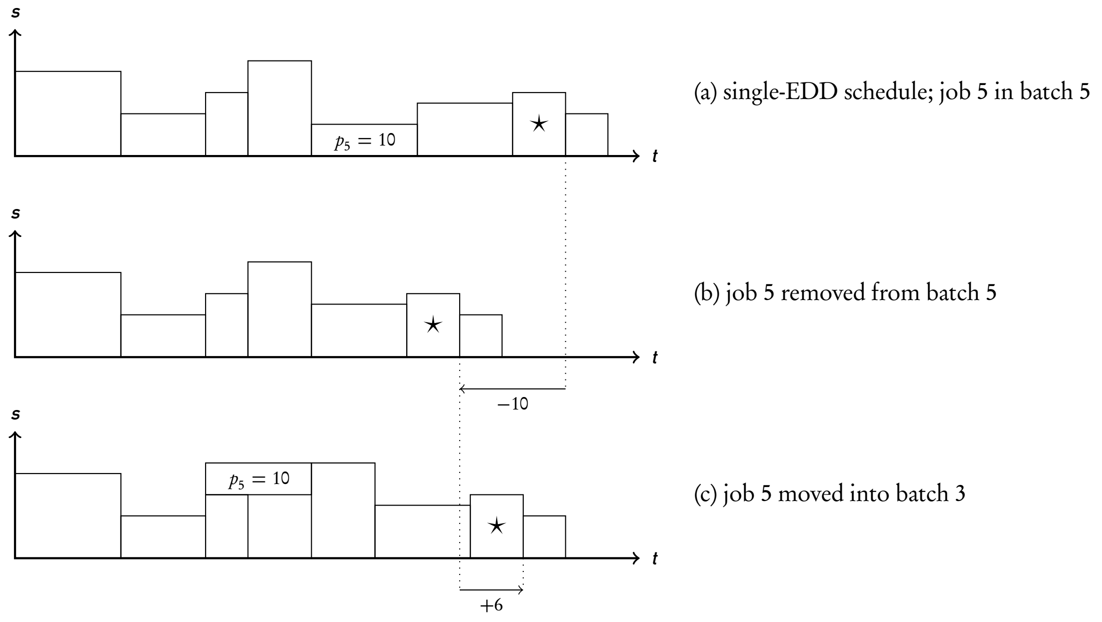
Consider now a single-EDD schedule. Moving j from its single-EDD batch k_j into an earlier batch k_α has the following effect:

- the lateness of all batches after k_j is reduced by p_j (Figure 4.4b),
- the lateness of all batches after k_α, including those after k_j, is increased by $\max(0, p_j - P_\alpha)$, where P_α is the processing time of batch α before j is moved into it (Figure 4.4c).

Since, in any batch k, only the host job's lateness (with index $j = k$) is relevant to $L_{\max}$, we can understand the lateness of batch k as the single-EDD lateness of job $k = L_{k,\text{single}}$, modified by summed effect all moves of other jobs into and out of earlier batches have on k, as listed above.

The following expression defines the lateness of a batch k based on this calculation:

$$L_k = L_{k,\text{single}} - \sum_{h=0}^{k} P_h - p_h(2 - x_{hh}) \quad \forall k \in K^\star \tag{4.52}$$



s
t
$p_5 = 10$
⋆
(a) single-EDD schedule; job 5 in batch 5
(b) job 5 removed from batch 5
−10
(c) job 5 moved into batch 3
+6

Figure 4.4: Moving a job in the move-based MIP. In this example, job 5 (marked "$p_5 = 10$") is moved into batch 3, which changes the lateness of job 7 (marked $\star$) from $L_{7,\text{single}}$ to $L_{7,\text{single}} - 10 + 6 = L_{7,\text{single}} - 4$.

where x_{jk} is 1 iff job j is assigned to batch k in the solution schedule. The sum term represents the effects that all moves have on previous batches up to, and including, k: $x_{hh} = 1$ iff job h is not moved from its single-EDD batch h, and so the sum term evaluates to $P_h - p_h$ for a batch in which the host was not moved. If h has guests, then $P_h - p_h$ is positive if any of the guests is longer than j_h (i.e. $P_h > p_h$); if not, or if h has no guests at all, it will be zero. Effectively thus, for batch h, the time $\max(0, P_h - p_h)$ is added to L_k. If, however, job h was moved out of its batch, then the batch will have a processing time of $P_h = p_h$ (the fixed lower bound for P_h). In this case, effectively, for batch h, the time p_h is subtracted from L_k. The net sum of these additions and subtractions to and from $L_{k,\text{single}}$ adjusts the lateness of batch k to its correct number given the values of x_{jk}.

Note also that we only need to calculate the lateness for $k \in K^\star$, the set of batches that are the last in their respective buckets; in other words, those with the longest processing time in their respective bucket. This is true because empty batches in a bucket will be reduced to length $P = 0$ by the minimization objective.

The full set of constraints is listed in Model 4.5.1, and the following section explains every constraint in detail. Constraints (4.55) and (4.54) are capacity constraints and uniqueness constraints: batches have to remain within capacity b, and every job can only occupy one batch.

Constraints (4.56) define the value of P_k for every batch k as the longest p of all jobs in k. This is required in the definition of batch lateness as described above and also in (4.58), which follows the explanation above.

Constraints (4.57) ensure that no job is moved into a host-less batch, i.e. in order to move job j into batch k ($x_{jk} = 1$), job k must still be in batch k ($x_{kk} = 1$).

Constraints (4.59) implement the requirement that jobs are only moved into earlier batches.

Constraints (4.60) apply to jobs with index greater than f_L, which is the index of the job with the largest value for $L_{k,\text{single}}$. All jobs after batch f_L can be ignored in the question, based on the proof given in Appendix ??. This constraint fixes those jobs to their single-EDD batches, effectively removing them from the model. This set of constraints is rarely active but can greatly decrease the size of the model in some instances.

Constraints (4.61) through (4.63) are dominance rules implemented as lazy constraints as defined in subsection 2.2.3, and are presented in greater length in the following subsections.

4.5.1 Symmetry-breaking rule

Constraints (4.61) implement the following concept: if two jobs j_1 and j_2 are assigned as guests to batches k_1 and k_2, and no other jobs (save for the hosts) are assigned to k_1 and k_2, then j_1 is always assigned to k_1 and j_2 to k_2. These constraints only hold for pairs of jobs where both p_{j_1} and p_{j_2} are less than both p_{k_1} and p_{k_2} (both batches are longer than the jobs) and all possible assignments are feasible capacity-wise.

Large problem instances generate thousands of such constraints, and only very few of them will ever hold in the model. Nevertheless, they can noticeably improve solving time

$$
\begin{aligned}
&\text{Min.} && L_{\max} && && (4.53)\\
&\text{s.t.} && \sum_{k} x_{jk} = 1 && \forall j \in J && (4.54)\\
& && \sum_{j} s_j x_{jk} \leqslant b && \forall k \in K && (4.55)\\
& && P_k \geqslant p_j x_{jk} && \forall \{j \in J, k \in K \mid j \geqslant k\} && (4.56)\\
& && x_{jk} \leqslant x_{kk} && \forall \{j \in J, k \in K \mid j > k\} && (4.57)\\
& && L_{\max} \geqslant L_{k,\text{single}} - \sum_{h=0}^{k} P_h - p_h(2 - x_{hh}) && \forall k \in K^{\star} && (4.58)\\
& && x_{jk} = 0 && \forall \{j \in J, k \in K \mid j > k\} && (4.59)\\
& && x_{jj} = 1 && \forall \{j \in J \mid j \geqslant f_L\} && (4.60)\\
&(*) && 2(4 - x_{k_1,k_1} - x_{k_2,k_2} + \sum_{\substack{j\\ j\neq j_1\\ j\neq j_2}} -x_{j_1,k_1} - x_{j_1,k_2} - x_{j_2,k_1} - x_{j_2,k_2}) \geqslant x_{j_1,k_2} + x_{j_2,k_1} && \begin{array}{c}\forall \{j_1, j_2 \in J,\\ k_1, k_2 \in K\\ \mid k_1 < k_2 < j_1 < j_2 \wedge\\ [p_i \leqslant k_h \wedge b - s_h \geqslant s_i\\ \forall i \in \{j_1, j_2\},\\ \forall h \in \{k_1, k_2\}]\}\end{array} && (4.61)\\
&(*) && 1 - x_{j_1,j_1} \geqslant x_{j_2,k} && \begin{array}{c}\forall \{k \in K, j_1 \in J, j_2 \in J\\ \mid k < j_1 < j_2\\ \wedge s_{j_1} \leqslant s_{j_2}\\ \wedge s_{j_2} + s_k \leqslant b\\ p_{j_1} \leqslant p_k \wedge p_{j_1} \geqslant p_{j_2}\}\end{array} && (4.62)\\
&(*) && 2 - x_{jj} - x_{kk} \geqslant \left(1.0 + b - s_j - \sum_{\substack{i=k\\ i\neq j}}^{n_j} s_i x_{ik}\right) / b && \begin{array}{c}\forall \{k \in K, j \in J\\ \mid j > k \wedge p_k \geqslant p_j\\ \wedge s_k + s_j \leqslant b\}\end{array} && (4.63)
\end{aligned}
$$

Model 4.5.1: Move-based MIP model. Constraints marked $(*)$ are lazy constraints (see subsection 2.2.3 for details).

in some cases; declaring them as “lazy” helps keep the model small.

4.5.2 Dominance rule on “safe” moves

For the following sections, note that a *safe* move is defined as one where the guest job is shorter than the host job ($p_{\text{guest}} \leqslant p_{\text{host}}$), i.e. a move that will under no circumstances increase the processing time of the host job.

This applies to pairs of jobs $j_1 < j_2$ that are, potentially, competing to be guests in batch k. If both j_1 and j_2 are shorter (in terms of p) than k, we can safely enforce that in some cases, j_1 gets priority over j_2, namely if both $p_{j_1} \geqslant p_{j_2}$ and $s_{j_1} \leqslant s_{j_2}$ hold. In this case, we can state that j_2 cannot be moved into k unless j_1 has been moved somewhere also (into k or some other host batch).

Two facts allow for this set of rules:

Proposition 3. *If a job j_h can either be safely moved into an earlier batch, or act as a host for later jobs $J_g = \{j_{g_1}, j_{g_2}, \dots\}$, moving j_h is always preferable.*

Proof. Let $k_{L_{\max}}$ be the batch hosting the job with maximum lateness in the final solution. If j_h is due after $k_{L_{\max}}$, moving it will have no impact on $L_{\max}$ (see Appendix ??). If j_h is due before $k_{L_{\max}}$, but the earliest-due of J_g is due after $k_{L_{\max}}$, then moving J_g into j_h will have no impact or a worsening impact on $L_{\max}$. If both j_h and J_g are due before $k_{L_{\max}}$, then moving J_g will, at best, improve $L_{\max}$ by p_{j_h} (if $\max_p(J_g) \geqslant p_{j_h}$). In this case, however, moving j_h back safely will, under any circumstances, reduce $L_{\max}$ by p_{j_h}. The first job in J_g can host all later jobs in J_g (and possibly more, since vertical capacity was freed). The total reduction in $L_{\max}$ in this case is improved. □

Proposition 4. *If a job j_1 with $d_{j_1} \leqslant d_{j_2}$ is both longer and thinner than j_2 ($p_{j_1} \geqslant p_{j_2}, s_{j_1} \leqslant s_{j_2}$), assigning it as a guest to k will produce a lower L_{max} than assigning j_2 to k.*

Proof. Because of due date and size requirements, j_1 is a potential host for j_2, Proposition 3 applies.

Generally, a longer guest job will result in a greater reduction in $L_{\max}$, and is thus preferable. Since j_1 is thinner than j_2, the dominance rule will not preclude (via capacity constraints) other guest jobs from being assigned to k as a consequence of prioritizing j_1 over j_2. □

4.5.3 Dominance rule on required safe moves

This constraint can be expressed logically as: if j can be safely moved into k without violating the capacity constraint, then j must be moved somewhere, or k must be empty (or both). In other words, in the context of Proposition 3, a schedule is unacceptable if job j is left in its single-EDD batch although there is room for it in an earlier batch.

The left side of the above *if-then* statement is written as $(b+1-s_j-\sum_{\substack{i=k\\ i\neq j}}^{n_j} s_i x_{ik})/b$, which evaluates to 1.0 or greater iff s_k plus the sizes of guest jobs in k sum to less than $b-s_j$.

Empirical comparison of models

5.1 Experimental setup

The models were tested on a set of job lists. Malapert's paper uses benchmark job lists by Daste et al. [2008], ?, with a capacity of $b = 10$ and values for p_j, s_j and d_j distributed as follows:

$$p_j = U[1, 99] \tag{5.1}$$

$$s_j = U[1, 10] \tag{5.2}$$

$$d_j = U[0, 0.1] \cdot \tilde{C}_{\max} + U[1, 3] \cdot p_j \tag{5.3}$$

where $U[a, b]$ is a uniform distribution between a and b, and $\tilde{C}_{\max} = \left(\sum_{j=1}^{n_j} s_j \cdot \sum_{j=1}^{n_j} p_j\right) / (b \cdot n)$ is an approximation of the time required to process all jobs.

Ten different sample job sets are used per unique value of $n_j \in \{10, 11, \ldots, 18, 19, 20, 25, 30, 35, 40, 45, 50\}$. The times shown in figure 5.1 are averaged over those ten instances for each n_j.

The models were run using Cplex 12.2 on an i7 Q740 CPU (1.73 GHz) in single-thread mode, with 8 GB RAM. Solving was aborted after a time of 3600 seconds (1 hour).

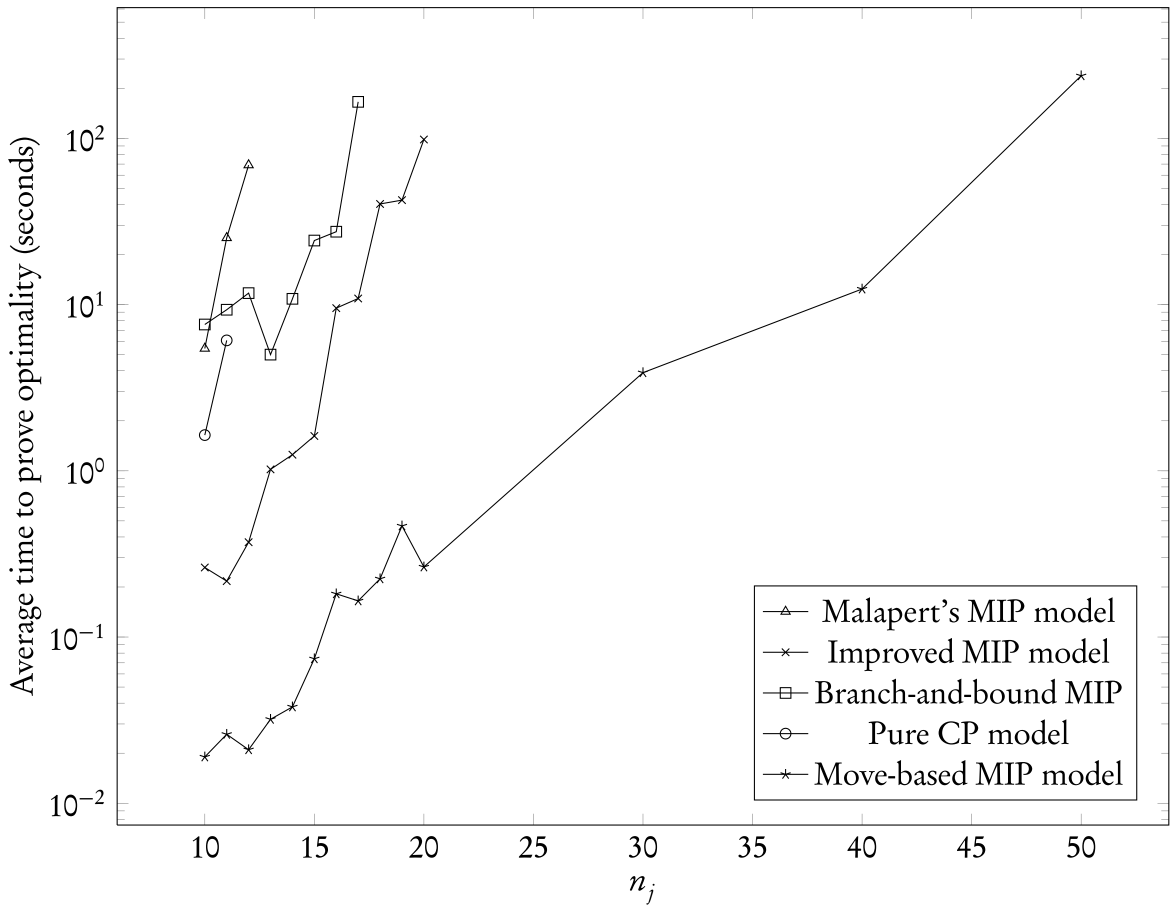


Figure 5.1: Comparison of CPU time used by different models to find an optimal schedule and prove its optimality.

5.2 Results

The CP branch-and-bound model times out on most instances and is not shown here. The pure CP model times out on one 12-job instance and gets progressively worse with more jobs; similar to Malapert's original MIP model.

The new, move-based MIP model trumps all other models' performance by at least one order of magnitude.

Unfortunately we have no such performance profile for Malapert's `sequenceEDD` global constraint. We do, however, have a reference in Figure 5.2, which shows the percentage of

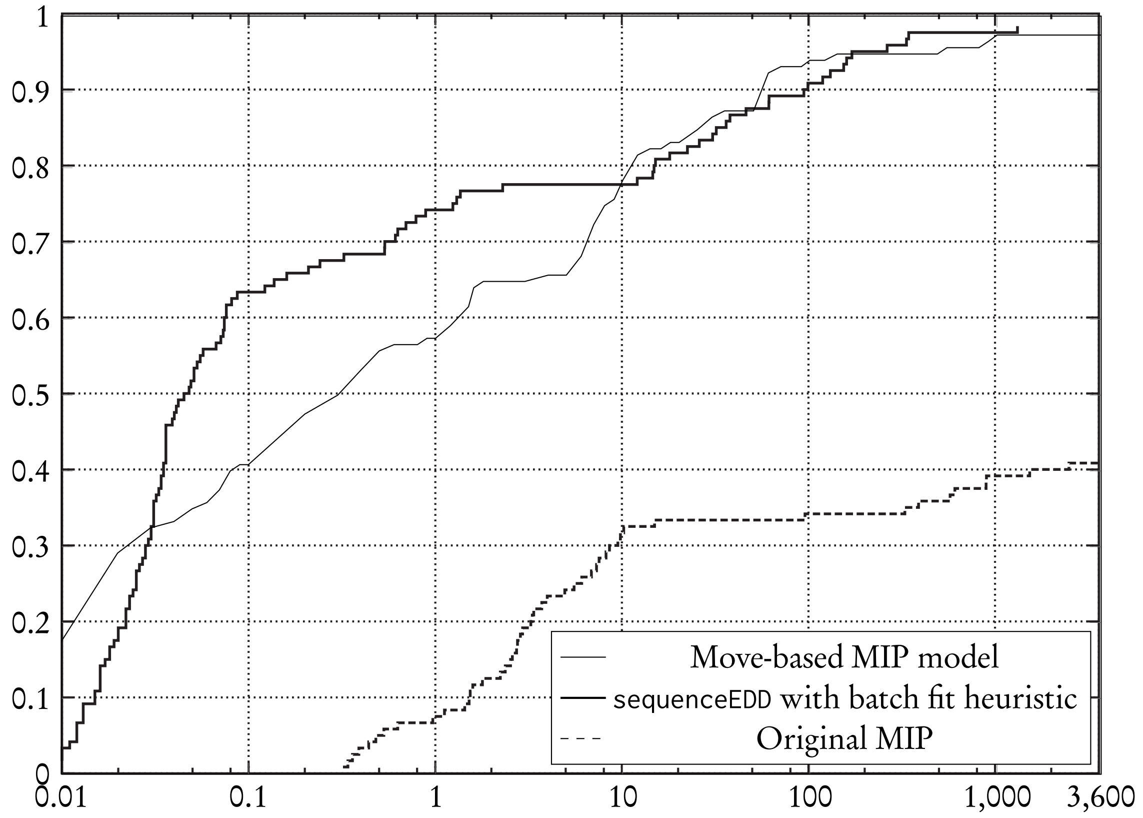


Figure 5.2: Percentage of problem instances solved in a given time

solved problem instances in a given time. The figure is based on $n_j \in \{10, 20, 50\}$, with 40 instances each (120 instances total).

5.3 Discussion

The move-based MIP model performs uses the fewest variables (columns) and constraints (rows) and performs the best. The use of lazy constraints for larger n_j is advantageous in the large majority of problem instances and thus warranted.

The improved original MIP is a great improvement on the original MIP, mostly owed to constraint (4.11) which excludes a large number of potential batch assignments.

The batch-by-batch decomposition using MIP to assemble batches, while an improvement over Malapert's MIP model, is not satisfactory; the relaxations are too lax for the decomposition to compensate for the overhead of a naive branch-and-bound search, since we can only exclude one failed solution at a time.

5.3.1 Comparison with Malapert's sequenceEDD global constraint

Using all four filtering rules in conjunction with a *batch fit* search heuristic allowed Malapert to solve 38 out of 40 problem instances with $n_j = 50$ in less than an hour on a cluster of Linux machines, each node of which using 48 GB of RAM and two quad core 2.4 GHz processors (that is, using considerably more powerful hardware than was used for this thesis).

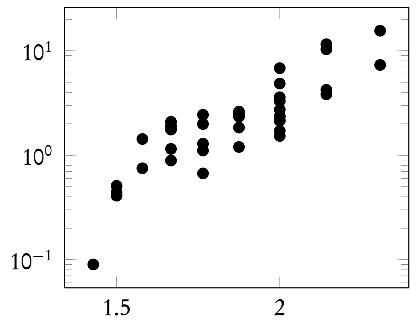
The move-based MIP model is able to achieve a comparable performance (39 of 40 instances solved within an hour). While it is, on average, slower on problems with low n_j (compare Figure 5.2), it is able to prove optimality quicker on most instances that take longer than 10 seconds to solve—ostensibly the more critical metric.

5.4 Further performance tests

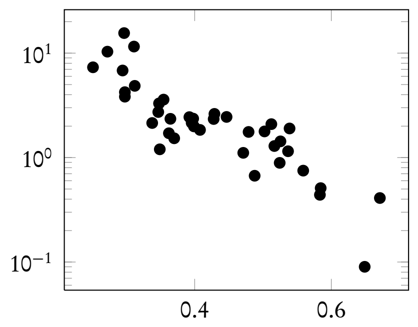
A better understanding of the relationships between model performance and problem instance characteristics can help in the development of faster models. This is especially true with regard to redundant and/or lazy constraints that are used only to achieve a tighter formulation at the price of a larger model: using such constraints is often warranted only for particularly challenging problem instances. The following paragraphs outline some such considerations.

5.4.1 Correlation between disjunctivity and solving time

The mean number of jobs per batch (n_j/n_k) in the optimal solution is strongly correlated with the time needed to solve to optimality. This is unsurprising given that a problem with a great number of large jobs (i.e. large s_j) will be of relatively low difficulty in terms of bin packing. This is illustrated by Figure 5.3a. An excellent predictor of this correlation appears to be what Baptiste and Le Pape Baptiste and Le Pape [2000] call *disjunction ratio*:



(a) n_j/n_k



(b) Disjunction ratio

Figure 5.3: Correlation between time to prove optimality in seconds (ordinates) and measures of disjunctivity (abscissae), based on 40 instances with $n_j = 30$. Subfigure (a) shows the average number of jobs per batch in the optimal solution. Subfigure (b) shows the *disjunction ratio* as defined in Section 5.4.1.

the ratio between the number of job pairs which cannot run in parallel (as $s_{j_1} + s_{j_2} > b$) and the total number of job pairs n_j^2.[1] The relationship is shown in Figure ??.

As a corrolary to the above, the solving time is greatly dependent on the capacity of the machine. Larger capacities, relative to the average job size $\bar{s}_j$, will cause longer solving times as the disjunction ratio decreases.

5.4.2 Uniform distribution of due dates

The due dates in the test instances used by Malapert and in the previous tests were generated by a fairly complex formula designed such that jobs are likely to fall into *buckets*, i.e. larger sets of jobs with equal due dates.

Malapert's `sequenceEDD` global constraint exploits the notion of buckets extensively, and would likely perform worse on instances that exhibit no bucketing at all (though we could not test this for lack of a working implementation).

Our move-based model uses only the latest-due batch in every bucket to define the value of $L_{\max}$. As a result, the size of the model is directly dependent on the number of buckets. We were, however, unable to find evidence of such a relationship: Figure 5.4 shows the

[1]The term originates from the dichotomy between *disjunctive resources* which can only process one job at a time, and *(highly) cumulative resources* which can process many jobs in parallel.

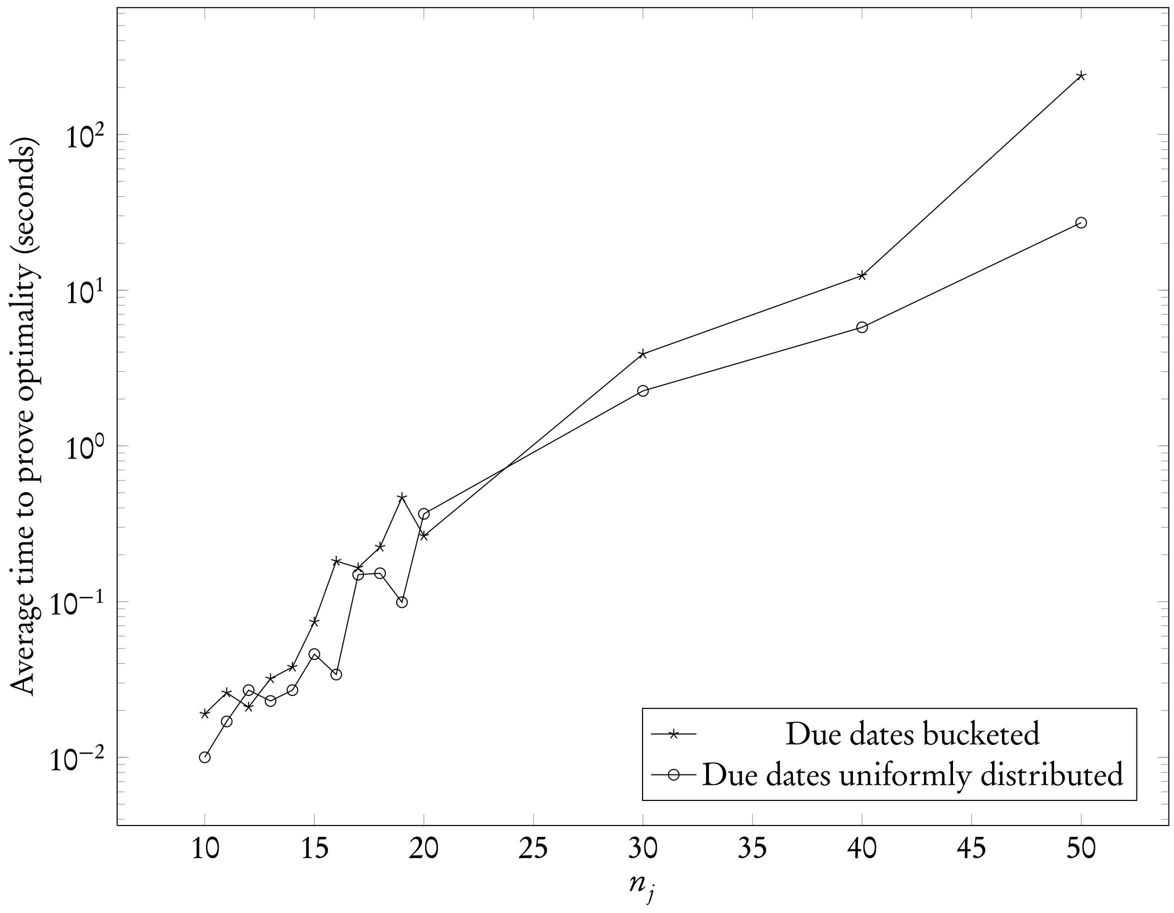


Figure 5.4: Comparison of time to prove optimality required by the move-based MIP model depending on the distribution of due dates.

performance of the move-based MIP on non-bucketed ($d_j = U[1,99]$) instances compared to bucketed instances.

Indeed, average performance improves with uniformly distributed due dates on instances with larger n_j.

Conclusion

The problem of scheduling jobs of non-identical sizes and processing times on a batch resource with a limited capacity arises frequently in the context of furnaces, autoclaves, driers, or other machinery where one particular job determines the processing time of the entire batch. The problem of minimizing the maximum lateness is given in situations in which all customers (or downstream processes) are to be treated as fairly as possible.

The problem was approached by Daste Daste et al. [2008] using a branch-and-price model and by Malapert Malapert et al. [2012] using a new `sequenceEDD` global constraint. In this paper we have introduced a set of improvements to Malapert's basic MIP model, a simple CP model, a decomposition method, and a new MIP based on the idea of independent search moves expressed in a single formulation.

The new move-based MIP model rivals the performance of Malapert's `sequenceEDD` constraint particularly with larger problem instances ($n_j > 30$). The great simplicity of our formulation allows production managers to integrate it into existing models without having to implement highly problem-specific filtering rules. Furthermore, it is easily extended with additional redundant or lazy constraints if necessary.

6.1 Future Work

All models described in this paper have weaknesses. The improved original MIP model lacks the fine-tuning and lazy constraints enjoyed by the move-based MIP. The CP model could likely benefit from additional redundant constraints. The decomposition approach needs a better pruning method. The move-based MIP's lazy constraints could certainly be fine-tuned based on a more generous definition of safe moves and conditional activation relying on the problem's *disjunction ratio*. Such potential improvements are described in Appendix **??**.

6.1.1 Application to related problems

The promising computational performance of the move-based MIP model might find application in similar problems, e.g. completion-time minimization on batch machines (which is equivalent to $L_{\max}$ minimization with equal due dates), or $L_{\max}$ minimization with release dates (i.e. coupling x_{jk} to batch start times, which can be expressed using an exact analogue to constraint (4.58)).

Bibliography

Meral Azizoglu and Scott Webster. Scheduling a batch processing machine with non-identical job sizes. *International Journal of Production Research*, 38(10):2173 – 2184, 2000.

Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.

Peter Brucker, Andrei Gladky, Han Hoogeveen, Mikhail Y. Kovalyov, Chris N. Potts, Thomas Tautenhahn, and Steef L. van de Velde. Scheduling a batching machine. *Journal of Scheduling*, 1(1):31–54, 1998.

Denis Daste, Christelle Gueret, and Chams Lahlou. A branch-and-price algorithm to minimize the maximum lateness on a batch processing machine. In *Proceedings of the 11th International Workshop on Project Management and Scheduling (PMS), Istanbul, Turkey*, pages 64–69, 2008.

Lionel Dupont and Clarisse Dhaenens-Flipo. Minimizing the makespan on a batch machine with non-identical job sizes: an exact procedure. *Computers & Operations Research*, 29(7):807 – 819, 2002.

Ronald Graham, Eugene Lawler, Jan Karel Lenstra, and Alexander Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

Ignacio E. Grossmann. Mixed-integer optimization techniques for the design and scheduling of batch processes. Technical Report Paper 203, Carnegie Mellon University Engineering Design Research Center and Department of Chemical Engineering, 1992.

Narendra Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

Ali Husseinzadeh Kashan, Behrooz Karimi, and S. M. T. Fatemi Ghomi. A note on minimizing makespan on a single batch processing machine with nonidentical job sizes. *Theoretical Computer Science*, 410(27-29):2754–2758, 2009.

Frédéric Lardeux, Eric Monfroy, and Frédéric Saubion. Interleaved alldifferent constraints: Csp vs. sat approaches, 2008.

Arnaud Malapert, Christelle Guéret, and Louis-Martin Rousseau. A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, 221:533–545, 2012.

Onur Ozturk, Marie-Laure Espinouse, Maria Di Mascolo, and Alexia Gouin. Makespan minimisation on parallel batch processing machines with non-identical job sizes and release dates. *International Journal of Production Research*, 50(20):6022–6035, 2012.

Christos H. Papadimitrou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1998.

M.T. Yazdani Sabouni and F. Jolai. Optimal methods for batch processing problem with makespan and maximum lateness objectives. *Applied Mathematical Modelling*, 34(2):314 – 324, 2010.

H. Paul Williams. *Model Building in Mathematical Programming*. Wiley, 1978.

Appendix: Potential improvements to the models

The following section details some redundant constraints, tighter variable bounds and potential fine-tuning opportunities that were not implemented in this papier, but may be of use in further research.

A.1 CP Model

A.1.1 Constraint on number of batches with length $P_k > p$

Since batches take on the processing time of their longest job, there is at least one batch with $P = \max_j p_j$. We can proceed to fill batches with jobs, ordered by non-increasing processing time, based on algorithm 2.

At the end of this algorithm, we can state:

$$\texttt{globalCardinality}(\{1,\ldots,n_j\},\{P_{k,\min},\ldots,P_{k-1,\min}-1\},P_k) \quad \forall k \in \{k_0,\ldots,k_{LB(n_k)}\}, \tag{A.1}$$

where the constraint takes three arguments (`cards`, `vals` and `vars`) and $P_{k,\min}$ denotes the minimum length of batch k.

The algorithm sorts jobs by non-increasing p, and then fills batches job by job. If a job fits into a previous batch, it is assigned there. If a job fits into multiple previous batches, it is assigned to the batch with the smallest remaining capacity. This is called *best-fit decreasing*

Algorithm 2 Generating lower bounds on batch lengths

```
$J^\star \leftarrow J$                                          ▷ initialize all jobs as unassigned jobs
$n_k \leftarrow 1$; $S_k \leftarrow \{0\}$; $P_{k,\min} \leftarrow \{0\}$       ▷ Create one empty batch of size and length zero
sort $J^\star$ by processing time, non-increasing
repeat
    $j \leftarrow J^\star$.pop()                                ▷ select job for assignment, longest job first
    loop through all $n_k$ existing batches $k$, first batch first
        $k_p \leftarrow \varnothing$                                   ▷ no feasible batch
        $c_{\min} = b$                                  ▷ currently known minimum remaining capacity
        if $s_j < b - S_k$ and $b - S_k < c_{\min}$ then
            $k_p \leftarrow k_p$; $c_{\min} \leftarrow b - S_k$
        end if
    end loop
    if $|k_p| = 1$ then
        $S_{k_p} \leftarrow S_{k_p} + s_j$                             ▷ assign job $j$ to batch $k_p$
    else
        if $n_k < LB(n_k)$ then
            $n_k \leftarrow n_k + 1$                                 ▷ open new batch
            $S_{n_k} \leftarrow s_j$; $P_{n_k,\min} \leftarrow p_j$            ▷ assign $s_j$ and $p_j$ to the new batch
        else
            leave the loop now and end.
        end if
    end if
until $J^\star$ is empty
```

rule, and works as follows: let $J^\star$ be the set of jobs sorted by p, then at least one batch will be as long as the longest job $j_1^\star$. If the next n jobs fit into this batch, then there is at least one batch not shorter than $j_{n+1}^\star$, and similarly for subsequent batches.

Unfortunately, the optimal solution may perform better than the packing heuristic in terms of "vertical" (s_j) bin packing, and may thus require fewer batches. We therefore need to find a lower bound $LB(n_k)$ on the number of batches, and we can only guarantee the first $LB(n_k)$ of the above constraints to hold in the optimal solution. Finding a true lower bound is a two-dimensional bin packing problem, which is NP-hard. A possible but naive lower bound is j_0, the number of jobs, ordered by decreasing s_j, that can never fit into a batch together.

A.1.2 Constraint on number of batches with length $D_k > d$

In a similar fashion, we can determine that the second batch must be due no later than the earliest-due job j_{m+1} that can *not* fit into the first batch – if we sort jobs by due date and fit the earliest m jobs into the first batch – and so on for subsequent batches. Once again, since best-fit decreasing may not perform optimally in terms of "vertical" packing, this may not be valid for batches beyond the known $LB(n_k)$.

A.1.3 All-different constraints on P_k and D_k

Furthermore, if all jobs have different processing times, all batches will have different processing times as well: `alldifferent`(P_k). If m out of n_j jobs have different processing times, we can still enforce `k_alldifferent`(P_k, m). Some work on `k_alldifferent` constraints has been done in [Lardeux et al., 2008].

Similarly, we knows that the constraint `k_alldifferent`(D_k, m) must be true if m out of n_j jobs have different due dates.

A.2 Decomposition approaches

A.2.1 Potential heuristics

Improve the initial $L_{\text{max,incumbent}}$ A better initial upper bound on $L_{\max}$ can help prune some branches of the search tree from the outset. There are several dispatch rules (or maybe other heuristics?) that could be explored to do this better.

Improve $L_{\text{max,incumbent}}$ during search It may be useful to use a heuristic like above to "complete the schedule" once a promising partial schedule has been generated. I have yet to identify situations where this is always helpful.

A.3 Move-based MIP model

A.3.1 Conditional activation of lazy constraints

The disjunction ratio measure introduced by Baptiste and Le Pape [2000] may be useful in conditionally activating or omitting certain lazy constraint sets, as it is known that they often have a net slowing effect on the solver when dealing with less difficult problems. This was not implemented for this paper and is thus not represented in the results in section 5.1, but it certainly has potential to bring down average solving times, particularly with smaller problem instances, for which Malapert's `sequenceEDD` global constraint currently beats our move-based MIP model.

A.3.2 Improvement of *safe move* definition

The concept of safe moves is used in several constraints. It describes moves in which a job j is moved into an earlier batch k and $p_j \leqslant p_k$.

The definition of safe can be extended to guest jobs j that fulfill the following condition: $p_j \leqslant LB(L_{\max}) + d_k - UB(S_k)$, where $LB(L_{\max})$ is a lower bound on the optimal value of $L_{\max}$ and $UB(S_k)$ is an upper bound on the start time of batch k. The term represents the longest processing time that batch k can be extended to without violating a known lower bound on the maximum lateness.

The a-priori computation of upper bounds on batch start times is challenging, and can only be based on capacity violations ("what is the longest combination of guest jobs that can be placed into batches up to and including k?"), which will likely make for relatively weak bounds. Nevertheless, a stronger notion of safe may be beneficial in some instances.

Appendix: Other approaches

B.1 Move-back search

The following is a proof referenced in Proposition 3, but it could easily be turned into a standalone local search method that finds good solutions quickly.

Proposition 5. *Begin with any EDD schedule (such as the single-EDD schedule). Moving any job j into an earlier batch k will never shift the position of L_{max} to the right (into a later bucket).*

Proof. In any partial schedule following EDD, let k be the batch with maximum lateness $L_k = L_{\max}$. If multiple batches have equal lateness $L_{\max}$, let L_k refer to the latest-due of these batches in the schedule. It has processing time p_k. Then the lateness of the batch before k must be $L_{k-1} \geqslant L_k - p_k$ as a consequence of the EDD sequencing. Any batch following k can have a lateness no greater than $L_k - 1$.

- Moving any single job from a batch following k into a batch before k will worsen $L_{\max}$, but not change its position. Such a move is never necessary to arrive at an optimal solution.
- Moving back any single job from a batch before k safely[1] will improve $L_{\max}$, but not change its position.
- Moving back a single pre-k job j from a batch β into an earlier batch α unsafely will reduce L_k by p_α; $L_{\max}$ may still be in k, or it may be found in any batch between (and including) α and β, since their lateness $L_{[\alpha,\ldots,\beta]}$ is now increased by $p_j - p_\alpha$.

[1]for the definition of "safe" and "unsafe" moves, compare section 4.5.2.

- If the max-lateness job j is single itself, it can be moved from k into an earlier batch α. If this is done safely, the batch immediately preceding k will still have a lateness $L_{k-1} \geqslant L_k - p_j$. All batches after k will have their lateness reduced by p_j, but since their maximum lateness did not exceed $L_k - 1$ before the move, it will now be at least 1 less than that of batch $k - 1$.
- If the max-lateness job j was moved back unsafely into a batch α, batches between and including α and $k - 1$ now have their lateness increased by $p_j - p_\alpha$, while batches after k have their lateness decreased by p_α. Again, batch L_{k-1} would exceed the maximum lateness of batches after k.

This shows that after a sequence of operations in which single jobs are moved into earlier batches, $L_{\max}$ will never shift to the right. □

In fact, this means that all jobs after the $L_{\max}$-job in a single-job EDD schedule can be ignored in the scheduling problem entirely, although this turns out to be quite inconsequential as that job is often at or near the end of the single-job EDD schedule, resulting from the fact that $L_{k-1} \geqslant L_k - p_k$. By the same token, high-quality solutions often have their $L_{\max}$ in an early batch.

B.2 Other attempts to improve performance

We present two approaches which, independently of the model used, lead to no change or a worsening of performance in all initial tests despite appearing very promising at first. They are mentioned here briefly only for the sake of completeness.

B.2.1 Upper bound on $L_{\max}$

An upper bound on $L_{\max}$ can be found a priori by using a dispatch rule to find a feasible, if not optimal, schedule. Algorithm 3 shows a variant in which the longest safely-fitting job is selected first. Using the "best-fit" heuristic proposed in Malapert's paper (see section 3.1.2) yields better bounds with other problem instances, but is not significantly better overall.

Algorithm 3 Dispatch rule to compute an upper bound on $L_{\max}$

```
Sort jobs $J$ by due non-decreasing due date
$x_j = 1 \forall j \in J$                                ▷ $x_j = 0$ iff job $j$ is a guest in an earlier batch
repeat Select next job $j$ from $J$
    if $x_j = 0$ then
        continue                                          ▷ skip loop and move to next $j$
    end if
    $J^* \leftarrow$ list of all jobs $i$ after $j$ where $p_i \leqslant p_j$
    Sort $J^*$ by non-increasing processing time and size
    repeat Select next job $i$ from $J^*$
        if $b - s_j - c_j \geqslant s_i$ then                ▷ if remaining capacity suffices for $i$
            $x_i \leftarrow 0$
            $c_j \leftarrow c_j + s_i$
            $B_i \leftarrow j$
        end if
        if $b \leqslant c_j + s_j$ then                      ▷ if capacity exceeded
            break
        end if
    until $J^*$ is empty
until $J$ is empty
$t \leftarrow 0$                                          ▷ time index
repeat Select next job $j$ from $J$
    if $x_j = 1$ then
        $t \leftarrow t + p_j$
        if $t - d_j > UB(L_{\max})$ then
            $UB(L_{\max}) = t - d_j$
        end if
    end if
until $J$ is empty
```

B.2.2 Bounding the number of batches n_k

Initially, the number of batches needed is assumed to be equal to the number of jobs: $n_k = n_j$. Reducing n_k by pre-computing the maximum number of batches needed shrinks the x_{jk} matrix, and prunes potential search branches in branch-and-bound decomposition approaches.

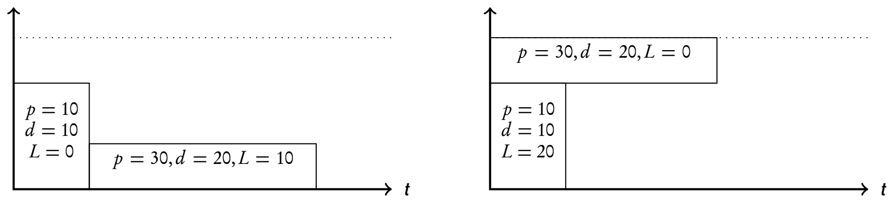


Figure B.1: Overzealous batch elimination can increase $L_{\max}$

Unfortunately, we cannot make a general statement that optimal solutions never have more batches than other feasible solutions – a simple counterexample is shown in figure B.1.[2]

Starting out with a one-job-per-batch schedule sorted by EDD, we can explore all feasible batch configurations recursively. To generate any other feasible schedule (including the optimal solution), jobs j are rescheduled ("moved back") from their original batch k_{origin} into a prior batch k_{earlier}. This eliminates k_{origin} and requires, of course, that k_{earlier} has sufficient capacity. If the job's processing time p_j exceeds that of k_{earlier}, then the lateness of batches between k_{earlier} and k_{origin} will increase. The merit of such a move cannot be judged a priori (unsafe batch elimination; compare). Safe eliminations, on the other hand, will never worsen $L_{\max}$, and only they can be considered when bounding n_k a priori.

Algorithm 4 outlines a recursive method to find an upper bound on n_k, recognizing safe batch eliminations only.

Algorithm 4 Recursive algorithm to find an upper bound on n_k

```
if no open jobs left then                                  ▷ if this is a "leaf node" in the recursion
    update UB($n_k$)
end if
find the combination of unsafe later jobs that fills up the capacity most, leaving us with capacity $b_r$
find all combinations $x$ of safe later jobs that fit into $b_r$
repeat
    $x$ = next safe job combination
    ignoreJobs ← $x$        ▷ make the moves, let the next recursion level deal with the rest of the jobs
    spawn and run child node with $J \setminus$ ignoreJobs
    ignoreJobs ← ignoreJobs $\setminus x$
until all combinations have been explored
return
```

This algorithm evidently requires exponential time. A relaxed variant is a possible option: if only a single unsafe move *into* a batch is possible, no safe eliminations into that batch are considered at all and we skip to the next batch. This would greatly speed up the recursion but also significantly weaken the usefulness of the resulting upper bound.

In a branch-and-bound decomposition approach in which batches are modelled individually, an upper bound on n_k could be used to limit the depth of the search tree, or, in

[2]To be more precise, we cannot state that at least one optimal solution is in the subset of feasible solutions that uses the fewest number of batches – a dominance situation that could be exploited, were it true.

combination with a method to determine a lower bound on the remaining jobs' n_k at every node, to actively prune the search tree during the search. The latter method, however, would also run in exponential time as it, again, would require knapsack-type reasoning unless we use a much less powerful relaxation.

Another application of a bounded n_k works as follows: a model (CP or MIP) is somehow optimized to quickly find the optimal solution of a problem that has a fixed number of batches $n_k \leqslant n_j$. The model is run with $n_k = n_j$, then with $n_k = n_j - 1$, then with $n_k = n_j - 2$ etc., until the value of $L_{\max}$ starts to increase (which happens at the point where an exceedingly low n_k forces capacity constraints to take precedence over optimality). With every successive run (i.e. with every decrement of n_k), the result of the previous run is passed to the solver as an initial solution to speed up the search—on the assumption that job-to-batch assignments will not vary substantially between runs.

We performed some such tests with a modification of Malapert's original MIP model. While passing initial solutions to the MIP solver greatly sped up the search for the optimal schedule at a given n_k, the overall time required to find the optimal solution to a problem instance was significantly greater. We therefore did not explore this approach further.

Appendix: Tabulated results

The following table lists solving time for all test instances.

n_j	i	t (seconds)
10	1	0.02
10	2	0.01
10	3	0.02
10	4	0.01
10	5	0.01
10	6	0.05
10	7	0.02
10	8	0.02
10	9	0
10	10	0.03
10	11	0.02
10	12	0.02
10	13	0.01
10	14	0.02
10	15	0.02
10	16	0.01
10	17	0.02
10	18	0.02
10	19	0.01
10	20	0
10	21	0.01
10	22	0.02
10	23	0.02
10	24	0
10	25	0.04
10	26	0.03
10	27	0.02
10	28	0.01
10	29	0.01
10	30	0.01
10	31	0.01
10	32	0.01
10	33	0.01
10	34	0.01
10	35	0.01
10	36	0.01
10	37	0.01
10	38	0.02
10	39	0.03
10	40	0.01
11	1	0.04
11	2	0.03
11	3	0.01
11	4	0.04
11	5	0
11	6	0.04
11	7	0.03
11	8	0.01
11	9	0.03
11	10	0.03
12	1	0.01
12	2	0.05
12	3	0
12	4	0.05
12	5	0.02
12	6	0.02
12	7	0
12	8	0.02
12	9	0.02
12	10	0.02
13	1	0.04
13	2	0.03
13	3	0.08
13	4	0.01
13	5	0.07
13	6	0.02
13	7	0.02
13	8	0.01
13	9	0.02
13	10	0.02
14	1	0.02
14	2	0.08
14	3	0.07
14	4	0.06
14	5	0.02
14	6	0.04
14	7	0.03
14	8	0.02
14	9	0.01
14	10	0.03
15	1	0.03
15	2	0.07
15	3	0.01
15	4	0.04
15	5	0.03
15	6	0.01
15	7	0.07
15	8	0.37
15	9	0.04
15	10	0.07
16	1	0.05
16	2	0.03
16	3	0.07
16	4	0.17
16	5	0.23
16	6	0
16	7	0.04
16	8	0.13
16	9	0.08
16	10	1.02
17	1	0.07
17	2	0.06
17	3	0.03
17	4	1.07
17	5	0.01
17	6	0.07
17	7	0.07
17	8	0.11
17	9	0.06
17	10	0.1
18	1	0.14
18	2	0.09
18	3	0.84
18	4	0.2
18	5	0.13
18	6	0.25
18	7	0.04
18	8	0.26
18	9	0.26
18	10	0.03
19	1	0.07
19	2	0.16
19	3	0.07
19	4	0.08
19	5	0.18
19	6	0.17
19	7	1.23
19	8	0.04
19	9	2.29
19	10	0.37
20	1	0.19
20	2	0.43
20	3	0.15
20	4	0.46
20	5	0.09
20	6	0.02
20	7	1.07
20	8	0.08
20	9	0.06
20	10	0.07
20	11	0.33
20	12	1.46
20	13	1.58
20	14	0.18
20	15	0.05
20	16	0.08
20	17	0.35
20	18	0.45
20	19	1.65
20	20	0.27
20	21	0.81
20	22	0.31
20	23	0.13
20	24	0.15
20	25	1.47
20	26	0.15
20	27	0.08
20	28	1.57
20	29	0.05
20	30	0.14
20	31	0.13
20	32	0.07
20	33	1.33
20	34	0.52
20	35	0.23
20	36	0.21
20	37	1.08
20	38	1.56
20	39	0.37
20	40	0.69
30	1	1.68
30	2	2.35
30	3	3.15
30	4	1.93
30	5	15.13
30	6	0.37
30	7	10.31
30	8	1.88
30	9	1.68
30	10	0.47
40	1	3.77
40	2	8.3
40	3	13.34
40	4	2.88
40	5	3.63
40	6	3.31
40	7	11.43
40	8	28.39
40	9	46.56
40	10	2.42
50	1	53.8
50	2	10.15
50	3	483
50	4	9.27
50	5	7.58
50	6	5.29
50	7	23.14
50	8	26.82
50	9	955.95
50	10	805.26
50	11	6.87
50	12	5.86
50	13	7.32
50	14	57.67
50	15	9.5
50	16	9.07
50	17	5.94
50	18	>3600
50	19	12.83
50	20	17.52
50	21	11.62
50	22	8.39
50	23	7.28
50	24	10.39
50	25	90.64
50	26	65.73
50	27	3.52
50	28	133.33
50	29	53.53
50	30	6.96
50	31	6.54
50	32	28.05
50	33	22.03
50	34	57.56
50	35	30.32
50	36	6.48
50	37	51.56
50	38	6.74
50	39	10.4
50	40	54.09

C.1 Results with disjunctiveness measure

n_j	i	t (s)	n_j/n_k	Disjunction ratio
30	1	1.84	1.875	0.407777777778
30	2	2.45	1.875	0.446666666667
30	3	3.3	2	0.347777777778
30	4	2.14286	2	0.337777777778
30	5	15.56	2.30769	0.296666666667
30	6	0.44	1.5	0.583333333333
30	7	10.33	2.14286	0.272222222222
30	8	1.99	1.76471	0.398888888889
30	9	1.78	1.6666	0.502222222222
30	10	0.51	1.5	0.584444444444
30	11	1.15	1.666667	0.536666666667
30	12	2.34	1.875	0.427777777778
30	13	1.9	1.66667	0.538888888889
30	14	2.73	2	0.346666666667
30	15	11.56	2.14286	0.311111111111
30	16	2.09	1.66667	0.512222222222
30	17	0.89	1.6667	0.524444444444
30	18	0.75	1.57895	0.558888888889
30	19	0.09	1.42857	0.648888888889
30	20	2.36	2	0.397777777778
30	21	1.76	1.66667	0.478888888889
30	22	2.35	2	0.364444444444
30	23	0.67	1.76471	0.487777777778
30	24	1.11	1.76471	0.471111111111
30	25	3.83	2.14286	0.297777777778
30	26	3.59	2	0.354444444444
30	27	7.32	2.30769	0.251111111111
30	28	1.71	2	0.362222222222
30	29	6.82	2	0.294444444444
30	30	0.41	1.5	0.671111111111
30	31	1.29	1.76471	0.516666666667
30	32	2.62	1.875	0.428888888889
30	33	2.15	2	0.395555555556
30	34	2.44	1.76471	0.392222222222
30	35	4.86	2	0.312222222222
30	36	1.53	2	0.37
30	37	4.23	2.14286	0.297777777778
30	38	1.43	1.57895	0.525555555556
30	39	1.2	1.875	0.348888888889
30	40	1.45	1.76471	0.362222222222